OpenMP Assignment

Joseba Hernández Bravo and Jorge Vicente Puig

November 2021

Parallelizing finite differences

We are going to exploit parallelism on the loops of the sequential program *finite_differences.c*.

Parallelization strategies

In this program we have three main loops that we can parallelize. For this purpose we will open a parallelize region at the begging of the program that includes the three loops: the inicialization loop, the main loop and the checksum loop.
The parallelize region is open using the following structure:

```
#pragma omp parallel num_threads(k) default(none) shared(U,L) +
+ private(x,t) reduction(+:S)
{
  /* Program that includes the three loops */
}
```

where k indicates the number of threads. In the following we will explain the use of the shared and private variables along with the reduction clause.

- **Inicialization loop:** These two loops are in charge of filling the matrix with the initial conditions. They can be parallelize with a *#pragma omp for* command:

```
#pragma omp for
for (x = 1; x < X; x++)
{
  U[x][0] = sin( x * M_PI / (double) X );
  U[x][1] = U[x][0] * cos( M_PI / (double) T );
}
#pragma omp for
for (t = 0; t <= T; t++)
{
  U[0][t] = 0.0;
  U[X][t] = 0.0;
```

```
}
```

For this parallelization we will have as privates variables x, t, in fact, they are counters for the loops so each thread must have a copy of them. The matrix U will be a shared variable.

We shouldn't expect a good performance using only this strategy, mainly because this loop is only executed once.

- **Main loop:** The main loop is the one in charge of doing the finite differences method. Althought is not possible to parallelize the outer loop (due to depence of iterations) we can parallelize the inner loop with the next commands:

```
for (t = 1; t <=T; t++)
{
    #pragma omp for
    for (x = 1; x < X; x++)
    {
        U[x][t+1] = (2.0 * (1.0 - L) * U[x][t] +
        + L * (U[x-1][t]+U[x+1][t]) - U[x][t-1]);
    }
}
```

In this loop, as in the previous one, x, t are private variables and U is still a shared variable along with L.

This parallelization should hardly reduce the computation time. In fact, we are paralleling a loop that would be done T times.

- **Checksum loop:** This loop will do a checksum of the final state. We need to use the reduction clause *reduction(sum: S)* to do a proper parallelism without losing any information.

```
#pragma omp for
for (x = 1; x < X; x++)
{
    S += U[x][T+1];
}
```

Performance analysis

In order to analize the performance we will do an analysis for the number of threads: 2,4 and 8, where we will compare the performance with different matrix sizes: $\{T = 10^4, X = i * 10^4\}$ and $\{T = i * 10^4\ , X = 10^4\}$ where $i = 1, \ldots, 5$. We include the execution time needed in our computer for each parallelization strategy.

We will compare the parallel results with the information provided in the following table, that includes the execution time depending on T and X for sequential programming.

Time in seconds for sequential programming with different sizes

T size	X size	Execution time
10000	10 000	3.384
20000	10 000	7.277
30000	10 000	10.503
40000	10 000	13.842
50000	10 000	17.788
10000	20 000	8.838
10000	30 000	16.029
10000	40 000	22.853
10000	50 000	29.818

There are included the execution time for different parallelization strategies in the following tables:

Time in seconds for parallelization strategies for $X = 10000$, $T = 10000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	1.860	1.787	1.773	1.792
4 threads	1.006	1.001	1.009	1.031
8 threads	0.928	0.924	1.007	0.950

Changing T size

Time in seconds for parallelization strategies for $X = 10000$, $T = 20000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	4.065	4.013	3.919	3.869
4 threads	2.377	2.287	2.303	2.298
8 threads	2.327	2.273	2.469	2.290

Time in seconds for parallelization strategies for $X = 10000$, $T = 30000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	5.477	5.614	5.613	5.593
4 threads	3.091	3.346	3.127	3.126
8 threads	3.010	3.069	3.063	3.062

Time in seconds for parallelization strategies for $X = 10000$, $T = 40000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	7.585	7.473	7.452	7.491
4 threads	4.169	4.179	4.174	4.177
8 threads	5.171	4.060	4.004	4.028

Time in seconds for parallelization strategies for $X = 10000$, $T = 50000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	9.168	9.539	9.391	9.422
4 threads	5.158	5.196	5.213	5.389
8 threads	4.968	5.118	4.998	5.078

Changing X size

Time in seconds for parallelization strategies for $X = 20000$, $T = 10000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	5.026	4.978	4.864	4.994
4 threads	3.001	3.012	3.043	3.032
8 threads	2.635	2.609	2.647	2.673

Time in seconds for parallelization strategies for $X = 30000$, $T = 10000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	8.823	8.778	8.844	8.950
4 threads	6.037	6.086	6.084	6.103
8 threads	5.214	6.566	5.301	6.633

Time in seconds for parallelization strategies for $X = 40000$, $T = 10000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	13.173	13.361	13.669	13.423
4 threads	9.489	9.270	9.356	9.247
8 threads	8.421	8.539	8.417	8.392

Time in seconds for parallelization strategies for $X = 50000$, $T = 10000$

Nº threads	All loops	Inizialization and main loops	Main loop	Main and checksum loops
2 threads	18.754	18.204	20.892	17.464
4 threads	12.886	15.403	12.846	12.936
8 threads	11.895	12.208	12.009	11.810

Conclusion

Firstly, we can conclude that there is an important reduction on execution time (around 50%) when applying any of the parallelization strategies for any of the X,T sizes, which make parallelization a performance strategy to take in count.

Focusing on the comparision of the different parallelization strategies, it is clear that **parallelizing all the loops is not always the best strategy**.
As expected it is necessary to balance the computation and communication time, so in some cases parallelizating the main loop leed as to better results.

Another important result to observe is that execution **time don't scale with number of threads**. In fact, the best performance improvement is done when using 2 threads instead of sequential programming and a significative improvement is done when increasing the number of threads to 4.
However, using 8 number of threads don't produce an important improvement in most of the cases, and even there are some cases where has the same results as using 4 threads.

Parallelizing 2D Laplace

In this second part of the assignment we will analize the *lap_ij.c* program that solves the 2D Laplace equation problem. We will implement different parallelization strategies in order to reduce the execution time.

Parallelization strategies

In this case, we have to handle a while loop that do 3 double for loops on each iteration. For that reason, we will open a parallel region using the next structure:

```
#pragma omp parallel num_threads(2,4 or 8) +
default(none) shared(pi,A,Anew,error,total_iter) private(i,j) firstprivate(iter)
  {
      /*Initialization part*/

      while ( error > TOL && iter < ITER_MAX )
      {
          /* Double for loops to parallelize */
      }
  }
```

In order to do a more proper parallelization, the functions will be removed and included in the main program, which will lead us to the following parallelization strategy:

```
//laplace_init
#pragma omp for
for (j=0; j<M; j++){
  A[    j    ] = 0.f;
  A[(N-1)*M+j] = 0.f;
 }
#pragma omp for
for (i=0; i<N; i++){
```

```
  A[   i*M   ] = sinf(pi*i / (N-1));
  A[ i*M+M-1 ] = sinf(pi*i / (N-1))*expf(-pi);
}

//laplace step
#pragma omp for collapse(2) schedule(static)
for ( i=1; i < N-1; i++ )
  for ( j=1; j < M-1; j++ )
    Anew[i*M+j]= stencil(A[i*M+j+1], A[i*M+j-1], A[(i-1)*M+j], A[(i+1)*M+j]);

//laplace error
error = 0;
#pragma omp for schedule(static) reduction(max: error)
for ( i=1; i < N-1; i++ )
  for ( j=1; j < M-1; j++ )
    error = fmax(error, sqrtf( fabsf( A[i*M+j] - Anew[i*M+j] )));

//laplace copy
#pragma omp for collapse(2) schedule(static)
  for (i=1; i < N-1; i++ )
    for (j=1; j < M-1; j++ )
      A[i*M+j]= Anew[i*M+j];
```

where we have to keep an eye on two clauses used: reduction and collapse.

The reduction clause is used in order to don't lose any information do to parallelization (as it was used in the checksum of finite differences), and have an error for the whole program instead of an error for each thread.

The collapse clause is used to distribute all the iterations done in a double for loop among all the used threads. Then, if we have an structure as these one:

```
for(i = 0; i < 4; i++)
    for(j = 0; j<100; j++)
        \* Some independent calculus */
```

then 400 iterations will be distributed independently along the threads. It is important to observe that it is completely necesary total independence between each iteration, condition that is satisfied in *Laplace Copy* and in *Laplace Step*.

Performance analysis

As we did in the previous section, we will make an analysis for each of the cases where 1, 2, 4 or 8 cores are used. In addition, we will compare the performance with different matrix sizes: $\{N = i * 10^3, M = i * 10^3\}$ where $i = 1, \ldots, 5$ and $ITER\_MAX = 1000$.

Time in seconds for sequential programming with different sizes

M size	N size	$ITER\_MAX$	Execution time (s)
1000	1000	1000	19,545
2000	2000	1000	70,400
3000	3000	1000	154,826
4000	4000	1000	269,206
5000	5000	1000	418,372

Time in seconds for parallelization strategies for $X, T = 1000$

Nº threads	All loops	Except inizialization
2 threads	10.084	10.006
4 threads	6.179	6.147
8 threads	5.375	5.326

Time in seconds for parallelization strategies for $X, T = 2000$

Nº threads	All loops	Except inizialization
2 threads	37.044	36.903
4 threads	23.641	23.800
8 threads	20.619	19.919

Time in seconds for parallelization strategies for $X, T = 3000$

Nº threads	All loops	Except inizialization
2 threads	79.955	80.499
4 threads	52.927	51.107
8 threads	45.009	44.814

Time in seconds for parallelization strategies for $X, T = 4000$

Nº threads	All loops	Except inizialization
2 threads	144.082	141.009
4 threads	98.573	91.330
8 threads	78.523	83.874

Time in seconds for parallelization strategies for $X, T = 5000$

Nº threads	All loops	Except inizialization
2 threads	216.865	236.018
4 threads	159.096	147.104
8 threads	140.345	129.918

Conclusion

First of all, we can conclude that the difference in execution time between the parallelised and sequential program is greater than 50%. In fact, depending on the number of threads used, this value can be up to three times less in the case of parallelized program.

On the other hand, we have that in the cases where 8 threads have been used, in general, we have obtained better results. However, we could say that the difference between the execution times obtained with 8 and 4 threads are not as significant as those obtained with 2 and 4 threads.

Finally, we have recalculated the execution times for the cases in which the initialisation of the program was sequential (the part in which the initial matrix is created). Thus, taking into acount the results obtained, we can conclude that the differences are not significant at all. In some cases, in fact, we have obtained better result using sequential programming. This is because it is more expensive to create the threads and run it in parallel than to just run it directly.